Lecture 16
RAII & Memory Management

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov (Northwestern), Hal Perkins (Washington), Godmar Back (Virginia Tech)

Administrivia

- Project specifications due today!
  - Each group gets will get feedback later this week

- In the meantime: get started on code now!
  - Start with the core functionality, which isn't really going to change

  - Due at the end of next week!!!
    - Worth more than double any homework assignment
    - Late penalty is significant. No slip days
      - Up to 90% max points for one day late
      - Up to 60% max points for two days late
      - Up to 30% max points for three days late

Lecture plan from here

- Thursday 5/25
  - C and C++ wrap-up
- Tuesday 5/30
  - Version control and Git
  - Also, the final quiz!
- Thursday 6/01
  - No lecture
  - Office hours in Tech Auditorium during lecture time

Today's Goals

- Consider the RAII programming idiom: (Resource Acquisition Is Initialization)
  - Understand how it is making development easier in C++
- Discuss C++ memory management
  - What exists and how it works
  - How to use Smart Pointers to make it easy too

Outline

- **C++ Strings**
- RAII
- C++ Memory Management
- Smart Pointers
- Example GE211 Project

Strings in C++

- Everything you wanted from C strings and didn't get

```
#include <string>


std::string s1 = "Test";
s1 += " String";
s1[0] = 'B';
std::cout << s1 << "\n"; // prints "Best String"
```

C++ string operations

- Iterators
  - Including reverse and constant
- Sizing
  - Characters and memory
- Access to characters

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return length of string (public member function)
length	Return length of string (public member function)
max_size	Return maximum size of string (public member function)
resize	Resize string (public member function)
capacity	Return size of allocated storage (public member function)
reserve	Request a change in capacity (public member function)
clear	Clear string (public member function)
empty	Test if string is empty (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Get character of string (public member function)
at	Get character in string (public member function)
back C++11	Access last character (public member function)
front C++11	Access first character (public member function)

https://www.cplusplus.com/reference/string/string/

C++ string operations

- Modification of strings
  - Add or remove from them
- Operations
  - Get C string from std::string
  - Find
  - Substring
  - Compare

Modifiers:

operator+=	Append to string (public member function)
append	Append to string (public member function)
push_back	Append character to string (public member function)
assign	Assign content to string (public member function)
insert	Insert into string (public member function)
erase	Erase characters from string (public member function)
replace	Replace portion of string (public member function)
swap	Swap string values (public member function)
pop_back C++11	Delete last character (public member function)

String operations:

c_str	Get C string equivalent (public member function)
data	Get string data (public member function)
get_allocator	Get allocator (public member function)
copy	Copy sequence of characters from string (public member function)
find	Find content in string (public member function)
rfind	Find last occurrence of content in string (public member function)
find_first_of	Find character in string (public member function)
find_last_of	Find character in string from the end (public member function)
find_first_not_of	Find absence of character in string (public member function)
find_last_not_of	Find non-matching character in string from the end (public member function)
substr	Generate substring (public member function)
compare	Compare strings (public member function)

https://www.cplusplus.com/reference/string/string/

Strings with different character sizes

- All are actually implementations of the generic `std::basic_string`
  - 16-bit “wide” characters
  - Strings of 8-bit, 16-bit, or 32-bit characters

Several typedefs for common character types are provided:

Defined in header `<string>`

Type	Definition
std::string	`std::basic_string<char>`
std::wstring	`std::basic_string<wchar_t>`
std::u8string (C++20)	`std::basic_string<char8_t>`
std::u16string (C++11)	`std::basic_string<char16_t>`
std::u32string (C++11)	`std::basic_string<char32_t>`

- UTF-8 mostly works with std::string by default
  - Some helper functions won’t work properly though…
  - Needs additional libraries for many functions

Outline

- C++ Strings
- **RAII**
- C++ Memory Management
- Smart Pointers
- Example GE211 Project

RAII-structured libraries enable simple dynamic memory

- std::vector, std::string, and other library containers must use dynamic memory internally
  - But we never have to call `vector.destroy()` or `free(string)`

- What makes memory management so automatic in C++?

- Programming paradigm: RAII
  - Resource Acquisition Is Initialization
  - Basic idea:
    - Wrap resources in an object
    - Allocate when you initialize and deallocate when destructed

What is a “resource”?

- Abstractly:
  - Something you need to get your computation done,
  - That you can run out of,
  - So you need to keep track of what you’re using and release what you aren’t

- Concretely:
  - Memory!
  - File handles
  - Network sockets
  - Database sessions
  - Acquired *locks* (concurrent programming)

The problem: leaking resources

```
#include <cstdio>

void handle_file(std:: string const& name) {
  FILE* f = fopen(name.c_str(), "r");

  // various code here using the file

}
```

Didn’t close the file!
There’s a resource leak!!

The problem: leaking resources

```
#include <cstdio>

void handle_file(std:: string const& name) {
  FILE* f = fopen(name.c_str(), "r");

  // various code here using the file

  if (some error occurred) { return; }

  // various more code using the file

  fclose(f);
}
```

What's wrong here?

The problem: leaking resources

```
#include <cstdio>

void handle_file(std:: string const& name) {
  FILE* f = fopen(name.c_str(), "r");

  // various code here using the file

  if (some error occurred) { return; }

  // various more code using the file

  fclose(f);
}
```

More common cause: early returns
Always beware when code returns early

Exceptions make early returns even worse

```
void helper(FILE* f) {
  if (some problem detected){throw std::runtime_error("Oops");}

  // various code here using the file
}


void handle_file(std:: string const& name) {
  FILE* f = fopen(name.c_str(), "r");

  // various code here using the file

  helper(f); // might throw an exception never "return"

  // various more code using the file

  fclose(f);
}
```

Can't clean up here without try/catch *everywhere*

C++ solution: Resource Acquisition Is Initialization

- Also known as Scope Based Resource Management (SBRM)

- Never open/close or free/allocate manually
- Instead make a class that owns the Resource
  - Allocate in the constructor
    - Programmer calls this when initializing the object variable

  - Deallocate in the **destructor**
    - Automatically occurs. Programmer doesn't have to do anything!

Destructors

- Same concept as constructors: used to clean up an object
  - Automatically called when the object goes out of scope
  - Note: you never call the destructor yourself!

- Handles any cleanup, including freeing necessary resources

```
std::ifstream::~ifstream() {
  // close the file here
}
```

Destructors allow resources to automatically be cleaned up

```
#include <fstream>

void handle_file(std:: string const& name) {
  std::ifstream f(name , "r");

  // do stuff with the file

} // f.~ifstream() happens automatically here
```

Destructors allow resources to automatically be cleaned up

```
#include <fstream>

void handle_file(std:: string const& name) {
  std::ifstream f(name , "r");
  // do stuff with the file.
  // Possibly return or throw exceptions!
} // f.~ifstream() happens here regardless
```

The destructor is guaranteed to run.
Even if there is an exception!

Break + Question

- In RAII (Resource Acquisition Is Initialization), when are resources allocated?

A. In the Constructor
B. By a special function, which is called on-demand
C. When compiling the code
D. Wherever **`new`** is called

Break + Question

- In RAII (Resource Acquisition Is Initialization), when are resources allocated?

A. In the Constructor

~~B. By a special function, which is called on-demand~~

~~C. When compiling the code~~

~~D. Wherever `new` is called~~

Outline

- C++ Strings
- RAII
- **C++ Memory Management**
- Smart Pointers
- Example GE211 Project

C++ memory management

- In C, dynamic memory was very important for making any realistic program that responds to user input

- In C++, because of RAII concepts and the Standard Template Library, we haven't had to manually use dynamic memory at all!
  - But it is still there, happening
  - And we could harness it ourselves if we need to

Reminder: C memory allocation

```
void* malloc(size_t size)
```

- Requests `size` bytes of memory from the heap
- Returns a pointer to this new **object**
  - Not associated with any variable (sort of like string literals)
  - It has no value by default
- The object persists until it is manually deallocated
  - Deallocated through a call to `free()`

C++ memory allocation (original, old style)

- Allocate with the `new` keyword and a type
  - No need to specify number of bytes anymore
  - Works for primitive types and for objects

  - Examples:
    - `int* value_ptr = new int;`
    - `Posn<int>* p = new Posn<int>;`

- Deallocate with the `delete` keyword and the pointer
  - Example: `delete p;`

- Warning: never mix-and-match malloc()/free() with new/delete
  - **UNDEFINED BEHAVIOR** (free() doesn't call destructor!!)

Dynamic arrays in C++

- For new, add the size of the array after the type

```
int* data = new int[10];
```

- For `delete`, must instead use `delete[]`
  - **Important**: Must remember this or **UNDEFINED BEHAVIOR** 😭
  - Reason:
    - `delete` calls the destructor and then frees the memory
    - `delete[]` iteratively calls destructors and then frees the memory
  - `delete[]` *could* have worked for everything, but it would be less efficient

Dynamic arrays in C++

- For new, add the size of the array after the type

```
int[10];
```

-

Just use std::array or std::vector instead!!

  - Reason:
    - `delete` calls the destructor and then frees the memory
    - `delete[]` iteratively calls destructors and then frees the memory

  - `delete[]` *could* have worked for everything, but it would be less efficient

C dynamic memory vs C++ dynamic memory

	`malloc()`	**`new`**
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	sometimes (often, but by a library without the dev knowing)
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a `void*` (*should be cast*)	appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	`free()`	`delete` or `delete[]`

Null pointers in C

- While NULL still works (legacy from C), there's a better way

- `nullptr` is the preferred literal
  - Same meaning as NULL, but its type is explicitly T* for *any* type T
  - Still converts to 0 when needed

  - C++ example:

```
void print(int* value_ptr);
void print(int value);

print(NULL); // calls print for type int 😱
print(nullptr); // calls print for type int* 😎
```

Outline

- C++ Strings
- RAII
- C++ Memory Management
- **Smart Pointers**
- Example GE211 Project

Using dynamic memory in a class

- Constructor will call `new` to allocate memory for some data member

- Destructor will call `delete` to free the memory when the object goes out of scope

- Observation:
  - Memory is manually created and initialized to values
  - But deletion is almost always just calling `delete`
  - We could use RAII to do this for us

C++ Smart Pointers (modern C++ memory management)

- A smart pointer is an object that stores a pointer to a heap-allocated object
  - Behaves just like a normal C++ pointer by overloading *, ->, [], etc.

- Smart pointers do the memory management for you
  - Automatically deletes the pointed-to object if the smart pointer goes out of scope
  - I.e., if the memory would leak, it is instead freed

- Smart pointers are the modern C++ way to do dynamic memory

Unique pointer (unique_ptr)

- Takes ownership of a pointer
- Allows access to the value pointed to
- Invokes `delete` automatically
  - Either when the `unique_ptr` goes out of scope via the destructor
  - Or when the owned pointer is overwritten

```
#include <memory>
std::unique_ptr<char> letter_ptr(new char('a'));

char letter = *letter_ptr; // sets letter to 'a'
```

Smart pointers don’t require new either

- Function `std::make_unique<TYPE>()` calls `new` for you
  - Totally gets rid of old C++ syntax (`new` and `delete`)
  - Pass in arguments for the Constructor as needed
  - Make it harder to mess up ownership

- Original way

```
std::unique_ptr<char> letter_ptr(new char('a'));
```

- True modern way

```
std::unique_ptr<char> letter_ptr = std::make_unique<char>('a');
```

Smart pointers are automatically freed

```
#include <memory>

void handle_memory() {
  std::unique_ptr<double> d = std::make_unique<double>(3.7);
  // do stuff with the pointer
  // Possibly return or throw exceptions!
} // memory is freed here regardless
```

The destructor is guaranteed to run.
Even if there is an exception!

Unique_ptr ownership rules

- Matches the "ownership rules" we discussed in C
  - There is only one single owner of a unique_ptr
    - Which in turn owns the memory
  - Cannot be copied

```
std::unique_ptr<int> x = std::make_unique<int>(5); // OK
std::unique_ptr<int> y(x); // Fails, no copy constructor
std::unique_ptr<int> z; // OK, holds nullptr
z = x; // Fails, no assignment operator
```

  - Ownership can be transferred if needed
    - release() gives up ownership of the pointer
    - reset() deletes the current pointer (if any) and stores a new one

Unique_ptr and arrays

- unique_ptr can store arrays as well
  - Will call delete[] on destruction

```
int main() {
  std::unique_ptr<int[]> x = std::make_unique<int[]>(5);
  x[0] = 1;
  x[1] = 2;

  return 0; // memory will be freed automatically
}
```

Shared pointers (shared_ptr)

- Similar to a `unique_ptr`, except that there can be multiple owners
  - Different ownership policy

- Tracks the number of owners to decide when to free
  - Copy/assign operators do work and increment number of owners
  - Destructor decrements number of owners
    - Frees memory if number of owners hits zero

- Technique is known as “reference counting”
  - Higher overhead than a `unique_ptr` has, means slower to use

Main takeaways

- Smart pointers are how memory is managed in modern C++
  - Never need `new` or `delete`
  - Never have to worry about creating the right amount of memory
  - Never have to worry about freeing the memory correctly

- `unique_ptr` automatically manages ownership rules for us
  - Ensures that there is only one owner at a time
  - Ensure that memory is properly freed if there would be no owner

Break + Question

- When is manual memory management necessary at all in C++?
  - Assumption: we’re using modern C++ with STL and smart pointers

Break + Question

- When is manual memory management necessary at all in C++?
  - Assumption: we're using modern C++ with STL and smart pointers
  - Need for very efficient code
    - Some kind of game engine that needs to reuse large amounts of memory
  - Implementing a new data structure from scratch
    - Some kind of crazy red-black tree nonsense
  - When something else needs to hold the memory for objects
    - Object slicing example from last lecture

Smart pointers fix object slicing

vector_of_base.hxx
vector_of_base.cxx

```
std::vector<std::unique_ptr<Printable>> heap;
heap.push_back(std::make_unique<Position>(1, 2));
heap.push_back(std::make_unique<Position3D>(-7, -6, -5);

for (std::unique_ptr<Printable> const& p: heap) {
  print_position(*p); // prints the right thing!
}
```

- Each object is heap-allocated, so there's enough memory for each
  - Pointers to each are inserted into the Vector
  - Objects are automatically freed when the Vector is destroyed

Outline

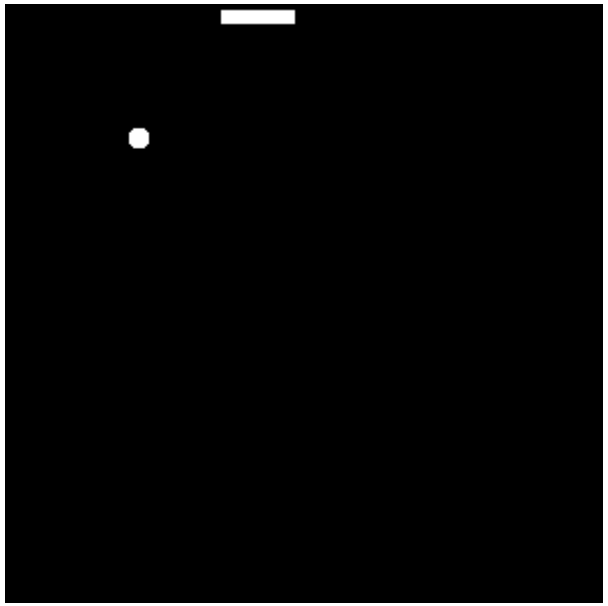
- C++ Strings
- RAII
- C++ Memory Management
- Smart Pointers
- **Example GE211 Project**

Multi-lecture project example

- Starting from https://nu-cs211.github.io/cs211-files/hw/final_project.zip

- We'll add features as we go
  - Probably not going to finish today
  - Plan to hop back into it in future lectures though

- Idea: Snake Game
  - Too simple for a final project
  - Simple enough to do in class?



Should the snake game be a grid-based game?

- Options:
  - Snake could be arbitrary `double` positions (like the ball in Brickout)
  - Snake could be fixed `int` grid positions (like the pieces in Reversi)

- Considerations
  - Hit detection: much simpler for grid than arbitrary positions

  - Movement: arbitrary physics or just move one grid position forwards

- Generally: much easier to make grid-based games
  - Pac-man, Tetris, and many others fit this same idea too!

Plan for game

- `List<Posn<int>>` for each "segment" of the snake
  - Consider the playing field as a 2D grid of locations
  - Posn<int> is one location on the grid

- Snake should "move" in current direction
  - Segment at end disappears
  - Segment at front gets added
  - Check for collisions
  - Occurs every N seconds?

- Draw each segment in the list to see the snake

- Key presses change direction of snake

Simplest initial design

- One segment only in the list

- Implement
  - Constructors
  - Model::on_frame() (most basic version)
  - View::draw()
  - Controller::on_key()

Start adding features

- Check for collisions
  - With edge of screen
  - With body of snake
- Goal object that increases snake length
- Obstacles to avoid
  - Get added randomly, or when eating food
  - But not right in front of the snake
- Resize draw based on screen dimensions and grid dimensions
  - Maybe even pick game dimensions at start

Outline

- C++ Strings
- RAII
- C++ Memory Management
- Smart Pointers
- Example GE211 Project